SCIENCE CHINA
Information Sciences

• RESEARCH PAPER •

August 2025, Vol. 68, Iss. 8, 182101:1–182101:19
https://doi.org/10.1007/s11432-023-3916-0

G-SPAC: a more granular greedy graph partition algorithm with spatial locality and judgment-aware edge folding

Yuedan CHEN[1], Zhijie LI[2,3], Guoqing XIAO[2,3*], Peixin XU[2,3], Xiaofei ZHANG[4] & Kenli LI[1,2]

[1]*Big Data Institute, Central South University, Changsha 410082, China*
[2]*College of Computer Science and Electronic Engineering, Hunan University, Changsha 410082, China*
[3]*National Supercomputing Center in Changsha, Changsha 410082, China*
[4]*Department of Computer Science, University of Memphis, Memphis 38152, USA*

Received 25 February 2023/Revised 3 August 2023/Accepted 30 October 2023/Published online 27 June 2025

Abstract Graph partitioning is the basis of many graph computations. A good graph partitioning algorithm is related to the partition balance and the communication overhead between nodes after partitioning. This paper measures the former by the relative standard deviation (RSD). At the same time, the latter is often related to the number of replicated vertices or edge cuts. In order to balance the above two goals, the greedy strategy is adopted to improve, which is a graph edge partitioning algorithm that transforms the graph so that the vertex partitioning methods can be applied to the transformed graph. A novel graph edge partitioning algorithm named G-SPAC is proposed. Firstly, to reduce the number of duplicated vertices, the method of degree-and-spatial-locality priority is designed so that adjacent vertices are more accessible to divide into the same subgraph. Secondly, to improve the division balance, the judgment-aware folding method is designed to carry out the graph division from a more granular perspective. Experiments are conducted on eighteen graph data sets. The results show that for most of the graphs, the RSD for the number of edges of each subgraph partitioned by the G-SPAC algorithm is below 3%. In terms of the number of vertex copies, G-SPAC's performance is not inferior to the SPAC algorithm, and it can even reduce the number of copies by an order of magnitude on some graphs.

Keywords graph edge partition, greedy method, spatial locality, judgment-aware folding, vertex partition

Citation Chen Y D, Li Z J, Xiao G Q, et al. G-SPAC: a more granular greedy graph partition algorithm with spatial locality and judgment-aware edge folding. Sci China Inf Sci, 2025, 68(8): 182101, https://doi.org/10.1007/s11432-023-3916-0

1 Introduction

The graph is a significant model showing how two or more sets of members are related. Compared with other models, the graph is much more expressive, playing an active role in many fields nowadays, such as social networks, recommended systems, and traffic networks. Take the core data of Facebook as an example here; its users could be represented as vertices, and relationships among users could be represented as edges so that a graph is constructed.

Being a flexible and expressive data model, graphs can be found in a wide range of applications, such as social networks [1–3], transportation networks [4, 5], computer vision [6–8], and even knowledge graphs [9–11] — the basis of the semantic web. Graph computing is an important research direction in academia and industry, and graph partitioning is one of the essential contents. In the era of big data, the growing graph scales impose higher requirements for graph computing capabilities and motivate various distributed parallel graph computing applications. As an essential prerequisite for realizing parallel and distributed computing systems, graph partitioning has attracted extensive research [12, 13].

Graph partitioning is to divide the graph into several modules. According to the number of divided modules, also known as k-way graph division. According to the ways of division, graph partitions can be categorized by vertex partition [14–21] and edge partition [22–31].

* Corresponding author (email: xiaoguoqing@hnu.edu.cn)

Vertex partitioning uses vertex separators in the execution of graph partitioning. McCrabb et al. [18] studied different vertex-pressure repartitioning schemes, which move vertices so as to co-locate them near their most relevant neighbors. Soudani et al. [19] proposed a graph partitioning approach based on the vertex-centric model that applies the personalized PageRank vectors of vertices and partitions to decide how vertices are joined partitions. Jiang et al. [20] introduced a new upper bound for the maximum k-plex problem, which is a partitioning of the candidate vertex set with respect to the constructing partial solution.

Compared with edge partitioning, vertex partitioning is more likely to cause load imbalance, especially for power-law graphs. In a power-law graph, most vertices have small degrees, and a few vertices have large degrees. When dividing by vertex, once a high-degree vertex is assigned to a partition, that partition's size will increase sharply, leading to an unsatisfactory division balance. Besides, edge partitioning saves more storage space than vertex partitioning due to the redundancy of edges [32].

An ideal graph partitioning strategy should minimize the partitioning overhead while maintaining a balanced partition. Partitioning overhead is twofold: time and space. Time overhead is the time cost to derive the partitions, where parallel methods are commonly used to improve time performance [33, 34]. Space overhead refers specifically to the additional overhead required in addition to storing a graph itself. For vertex partitioning, the extra overhead in space is the storage of redundant edges; for edge partitioning, the overhead is the storage of redundant vertices. The redundant structure also represents the connection between the partitions. Less redundancy means fewer inter-partition connections and less communication. However, obtaining the optimal solution for partitioning has been shown to be an NP-hard problem [22] because achieving minimum cut ratios and maximum load balancing is non-trivial.

Meanwhile, it may be challenging to make this ideal strategy when dealing with power-law graphs. In the real world, most of the graphs are power-law graphs [35], which means that most of the vertices have a few neighbor vertices while a few vertices are connected to a large number of vertices. The power-law feature of natural graphs brings many challenges. Usually, it will lead to the workload imbalance, the "curse of the last reducer", where 99% of the computation can be done quickly, while the others take much longer time to finish [36]. Besides, it is challenging for communication since it may cause communication asymmetry [37]. Furthermore, the skewed degree distribution makes splitting the graph much more difficult [27, 38].

At present, there are two kinds of graph partition models: one is based on stream, and the other is based on subgraph. Gonzalez et al. [37] proposed two simple edge partitioning methods. These two methods both distribute edges by scanning every edge in a linear way. One of the methods, referred to as random methods, assigns edges to each partition at random. The other method, the greedy method, prefers partitions where an end vertex of the edge to be allocated belongs. If no such partition exists, the partition with the least number of edges is selected. A balanced edge partitioning method is proposed by Bourse et al. [32] based on weighted vertex partitioning (WVP). In this method, weights are assigned to each vertex whose degree is exactly the value of its weight. The weighted graph is divided in balance according to the total weight of the vertices. Next, if the endpoint of an uncut edge falls into the ith vertex partition, it is assigned to the corresponding i-th edge partition. For edges that are cut, they are distributed randomly or greedily to reach a partitioning balance. However, both of the above methods are stream-based with poor partitioning quality, which is far inferior to the SPAC model [39].

Subgraph-based partitioning is such as hypergraph partitioning models [40]. A hypergraph is a generalized graph whose edges are hyperedges, meaning that one edge may connect not only two vertices. In [40], vertices are interpreted as tasks, while hyperedges are interpreted as data objects, covering all tasks working with the data object. When the task is divided into k parts, minimizing the copies of data in memory corresponds to minimizing hyperedge cutting. This method can obtain better partition quality, and less data replication. On the other hand, the SPAC model [39] provides similar partitioning quality while being more scalable.

The SPAC [39] is a graph edge partitioning algorithm that transforms the graph so that the vertex partitioning methods can be applied to the transformed graph to obtain the division scheme from which the edge division of the original graph can be derived. However, SPAC's performance can be unsatisfactory due to the following constraints.

• Division balance is essential when evaluating graph partitioning methods. SPAC uses the vertex partition method, e.g., METIS [41], on the converted graph. The partition balance in METIS is defined as km/n, where k is the target number of partitions, m is the size of the largest subgraph, and n represents the vertex number of the original graph. So, there is no evidence to indicate that this definition is also

applicable to SPAC.

• SPAC does not make full use of the distribution law of graphs and may lead to unnecessary vertex splitting. Experimental results show that cutting such vertices is highly efficient since large-scale edges are partitioned by cutting a few vertices. This inspires us to use a greedy strategy to improve the algorithm.

In this study, we propose the G-SPAC algorithm and compare the performance of the algorithm in terms of partition balance and partition cost. Since the critical challenge to graph edge partition is seeking balanced partition output while minimizing the copies of vertexes, comprehensive measures are applied in our solution. G-SPAC performs spatial locality priority division at a more fine-grained perspective and uses the edge folding process with judgment conditions to approach this challenge. To summarize, the main contributions of this work are as follows.

• To reduce the vertex cuts, we adopt a greedy strategy to improve the SPAC algorithm that leads to cutting vertices with large degrees.

• To reduce the cost of division, we design the degree-and-spatial-locality-prior method introduced in Subsection 6.2 so that subgraphs close to each other are more likely to be divided together.

• To alleviate the imbalance of division, we propose judgment-aware folding where graph partitioning is conducted while folding edges, allowing graph partitioning in a finer granular perspective.

• We used 18 datasets from real-world sparse graphs, DIMACS[1)], and BHOSLIB[2)]. As shown in the experimental results, the division balance is less than 3% for most datasets. G-SPAC demonstrates excellent performance for mesh-like graphs in reducing the cost of vertex-cut by an order of magnitude compared with the SPAC algorithm.

The rest of our article is organized as follows. We present the background knowledge in Section 2. The related work is reviewed in Section 3. Section 4 describes the problem definition. Section 5 gives an overview of the algorithm. Section 6 explains each part of the proposed algorithm in detail. Performance evaluations are carried out in Section 7. Finally, Section 8 concludes this paper and outlines future directions.

2 Background

In the age of data explosion, with the explosive growth of data, the size of the graph increases dramatically, bringing complexity when analyzing such a graph. Therefore, a distributed system was born to address such a problem, and a graph partition technique is required to split the graph into several partitions and deliver them to a cluster of computing nodes. Load balance and minimal communication cost are important indicators when dividing a graph. Regarding load balance, the amounts of graph data in each computer node must have little difference, so the "curse of the last reducer" could be broken. As for communication cost, it is derived from the edges/vertices across different compute nodes, ensuring the synchronization among all nodes. Unfortunately, it has been proven an NP-hard problem to divide a graph evenly and minimize the number of vertex copies at the same time [22]. In our paper, both of the targets will be examined.

There are mainly two ways to partition graphs: vertex partition and edge partition, as shown in Figure 1.

2.1 Vertex partition

In vertex partition, vertices are assigned to the computing nodes without replication, so each vertex exists only in one node. The edges across nodes will be cut and stored by the nodes the two end vertices belong to. In this method, the cost of storage and communication is derived from the cut edges. When splitting a natural graph, vertex partition may lead to severe workload imbalance because of the power-law degree distribution.

2.2 Edge partition

Oppositely, edge partition is designed to assign edges to the computing nodes, and each edge is forbidden to cross nodes. Therefore, the split vertices connect subgraphs, and the cost depends on the total number of vertices among all the computing nodes. Compared with the vertex partition, the edge partition is

1) DIMACS. Dimacs challenge. http://dimacs.rutgers.edu/Challenges/.

2) BHOSLIB. Benchmarks with hidden optimum solutions for graph problems, 2004.

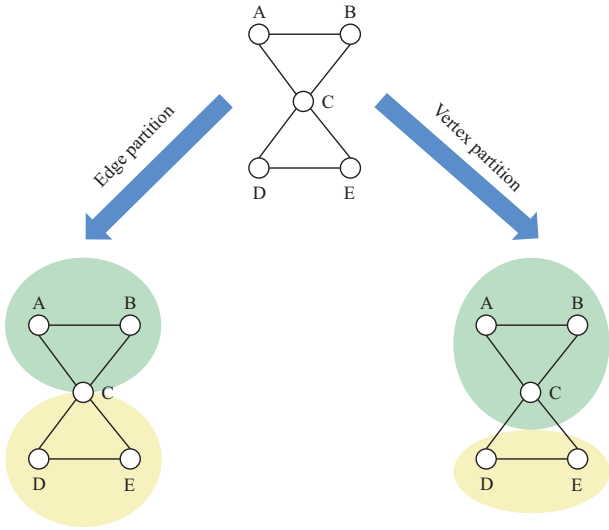


Figure 1 (Color online) Edge partition and vertex partition.

more efficient, providing a good quality of workload balance. This is because the overhead of computing the graph is substantially up to the edge count instead of vertex count [42].

3 Related work

On the one hand, graph division needs to reduce the division overhead; on the other hand, it needs to achieve load balancing. But achieving both at the same time proved to be NP-hard [22]. In an attempt to enhance the graph partitioning algorithms' performance, some researchers have proposed different methods from various aspects [22–26].

3.1 Partition cost

Regarding the edge partition model, the partition cost refers to the number of replicated vertices, which not only affects the memory cost but also the communication cost between partitions. Therefore, minimizing the number of vertex-cuts is an important research content of the graph edge partitioning algorithm.

A real-time edge re-partitioning algorithm was proposed by Mayer et al. [26] to deal with dynamic graphs, reducing the number of duplicate vertices by reducing unnecessary migrations in the re-partitioning process. For dynamic graphs, it is necessary to deal with the migration of edges; sometimes, edges are migrated as soon as they are assigned. That is unnecessary migrations. In this method, according to the distribution of the neighbors of the new edge, it is distributed to the most likely partition after re-partitioning. Consequently, the migration overhead is reduced, and the graph partitioning performance is improved.

Li et al. [25] proposed a new hybrid edge partitioner (HEP). HEP divides the edge set into two subsets: one is divided by the efficient memory algorithm NE++, and the other is divided by the flow method. The HEP method reduces the memory overhead by flexibly using these two subsets. Zhang et al. [22] proposed an edge partition heuristic algorithm, neighbor expansion (NE), which is a partition model based on a greedy strategy to maximize edge locality.

Unlike the above methods, we propose the G-SPAC algorithm to reduce the number of replicated vertices in this work. This is an edge partitioning model that utilizes graph transformations and greedy methods. At the same time, the character of spatial locality of graphs is under consideration, and the degree-and-spatial-locality-prior method is used to reduce the graph partitioning overhead.

3.2 Load balancing

Load balancing is also an important goal of graph partitioning algorithms. If the load is unbalanced, the calculation of the heavily loaded partition often becomes the bottleneck, limiting the calculation of the entire graph.

Sheng et al. [30] proposed GraBi — a partitioning framework for bipartite graph with communication efficiency and workload balance, comprehensively utilizing the bipartite graph structure to improve the overall performance. The degrees of vertices within each subset of bipartite graphs tend to be highly skewed. To address the imbalance problem of workload which results from the deviation of vertex degrees in each vertex subset, each large-degree vertex block is horizontally decomposed into several sub-blocks under an upper bound on the number of edges, and they are distributed across nodes in balance by a set of hash functions.

Bourse et al. [32] proposed an approximation algorithm for the problem of balanced edge partitioning with and without message aggregation. They describe the expected costs of vertex and edge partitioning with and without message aggregation and utilize them in common strategies for uniformly random placement of vertices or edges into one of the partitions. The algorithm matches the best-known approximation ratio for the balanced vertex partitioning problem without aggregation and shows that this still holds true for cases where the aggregation factor is equal to the maximum degree of the vertex. The implementation of load balancing is also considered in our method, and a judgment-aware folding method is proposed for this purpose.

3.3 SPAC algorithm

SPAC algorithm [39], known as split-and-connect algorithm, is an edge partition algorithm via expanding the graph and conducting vertex partition method, for example, METIS [41], on the expanded graph. Eventually, the original graph is split according to the plan of the last step. It is a simple yet efficient algorithm. In [39], experiments were conducted to find that the number of vertex copies of WVP [32] model is far more than the SPAC algorithm, and its average degree of the split-vertices is much less than the latter. It is inferred that the WVP model tends to avoid cutting vertices with large degrees. However, cutting such vertices has high efficiency since large scale of edges are partitioned via cutting a few vertices.

Inspired by the experimental results, we devise a greedy method on top of the SPAC model, intending to avoid cutting the small-degree vertices. The result is compared with the opposite version.

4 Problem definition

This paper focuses on finding a partition model to balance subgraph partition and reduce communication overhead. As mentioned above, for the edge partition model, the former needs to keep the number of edges in each subgraph the same, while the latter needs to reduce the number of vertex segments as much as possible. However, it is an NP-hard problem to balance the subgraph partition and reduce communication overhead simultaneously.

It is noted that the SPAC model has a good performance in reducing the partitioning overhead, and its experimental results indicate the direction of model optimization — partitioning the vertices with great degrees can improve the partitioning efficiency. In order to reduce the number of vertex segments and the communication overhead of the subgraph, an improved model of SPAC is proposed to cut vertices with great degrees using a greedy strategy. In this paper, the cost of vertex copies among subgraphs is a vital index to value the communication overhead of an edge partition algorithm. Carefully, the first copy of each vertex is not contained in this cost since it is essential to the graph.

Definition 1 (Vertex copies)**.** A graph is defined as $\boldsymbol{G} = (\boldsymbol{V}, \boldsymbol{E})$, in which $\boldsymbol{V}$ is the vertex set, and $\boldsymbol{E}$ is the edge set. Let n represent the number of vertices in $\boldsymbol{V}$, and m represent the number of edges in $\boldsymbol{E}$. Then, $\boldsymbol{G}' = (\boldsymbol{V}', \boldsymbol{E}')$ refers to the graph transformed from $\boldsymbol{G}$. In the new graph $\boldsymbol{G}'$, n', the size of the vertex set $\boldsymbol{V}$, equals double the size of the edge set of $\boldsymbol{G}$, that is $n' = 2m$.

Let $\boldsymbol{C}$ represent the vertex copies we have mentioned above; then it can be calculated by

$$C = \sum_{i=1}^{k} n_i - n, \tag{1}$$

where n_i represents the amount of vertex in the ith partition. Therefore, the cost of vertex copies is the sum of the unessential copies for each vertex, as (1).

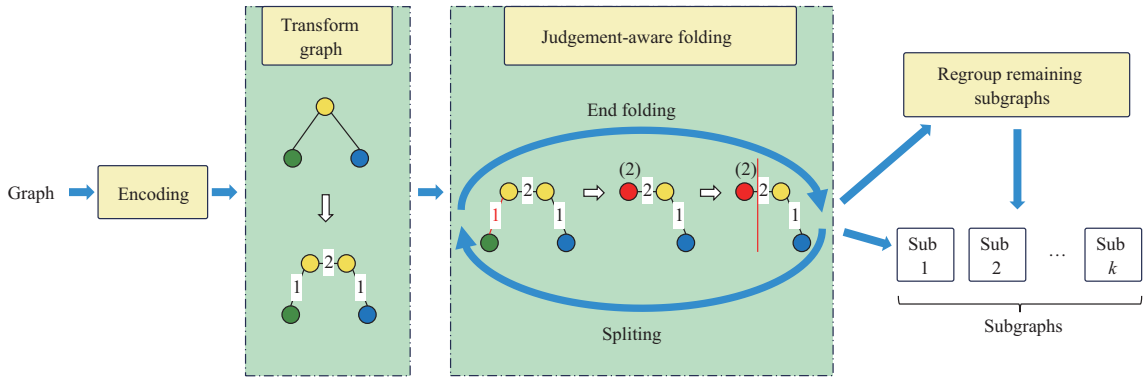


Figure 2 (Color online) Overview of the G-SPAC algorithm.

Generally speaking, the input to a graph is locally ordered. Using this spatial locality, it is possible to draw adjacent edges into the same subgraph as much as possible, thus further reducing the number of vertex copies produced by partitioning.

In addition to the communication overhead, the implementation of partition balance is also important. Unbalanced partitioning results in overburdened individual machine nodes, where data processing is heavily congested while other lightly loaded machine nodes sit idle. It wastes resources and seriously reduces the efficiency of data processing. In order to solve the partition balance problem, the judgment-awarded folding method is proposed. That is, the scale of the subgraph is judged while folding an edge to provide a fine-grained partitioning response. In this paper, the relative standard deviation (RSD) of the scales of subgraphs is used to measure the balance of the partition results.

Definition 2 (RSD)**.** RSD is the relative standard deviation of a set of numbers, where each number represents the number of edges of a subgraph (also named the scale of the subgraph). To be precise, RSD is the ratio of the standard deviation of scales and their average. So it can be expressed as

$$\mathrm{RSD} = \frac{\mathrm{SD}}{\bar{x}}, \tag{2}$$

where x is the ideal scale of each subgraph.

5 Algorithm overview

The general overview of the algorithm is shown in Figure 2. The graph, as input to the algorithm, is divided into several subgraphs through a series of processes. To begin with, there is an encoding step to encode the graph and build the relevant indexes. After the preparations are done, the core part of the algorithm begins.

Broadly speaking, when splitting a graph with the G-SPAC algorithm, there are three steps, as shown in algorithm pseudocode given by Algorithm 1. First, graph $\boldsymbol{G}$ is converted to graph $\boldsymbol{G}'$. Second, edges of $\boldsymbol{G}'$ are folded until it is up to the limit, and relevant edges are split from graph $\boldsymbol{G}$. Third, the remaining pieces are regrouped to form other partitions.

Step 1: transformation. The first step is to transform the graph. Specifically, it shows the internal structure of the vertex, where any edge connected to the vertex corresponds to one of the internal vertices, and these internal vertices are also connected. Detailed instructions are placed in Subsection 6.2. When transforming the graph, a priority queue is established to store the edges, and all the edges of the transformed graph are inserted into the queue so that these edges are sorted by weight in the queue. This step will be described in detail in Subsection 6.2.

Step 2: judgment-aware splitting. The second step is to fold the edges in loop. It folds one edge at a time from the priority queue. The specific meaning of folding will also be introduced in Subsection 6.3. After folding, the subgraph formed by the regions connected by the edge is obtained. Determine whether the size of the new subgraph exceeds the limit. If not, take another edge from the priority queue and repeat the above operation.

If the size of the new subgraph meets the requirements, it means that dividing the subgraph can be carried out. The logic is to split this subgraph from the original image, but in practice, you only need to

Algorithm 1 G-SPAC.

Require: G, k;
Ensure: $g_1, \ldots, g_k$.
// This is the overview of G-SPAC.
// 1. Transform graph.
upload(G);
$G' \leftarrow$ TransformGraph(G);
$h \leftarrow$ buildHeap(G');
$L \leftarrow G.m/k$;
// 2. Judgment-aware splitting.
while $i \leqslant k$ **do**
  $e \leftarrow h$.top();
  h.pop();
  $g \leftarrow$ folding(e);
  if g.cnt $\geqslant L$ **then**
    $g_i \leftarrow$ split(g);
  end if
end while
// 3. regroup.
regroup(h);
return $g_1, \ldots, g_k$.

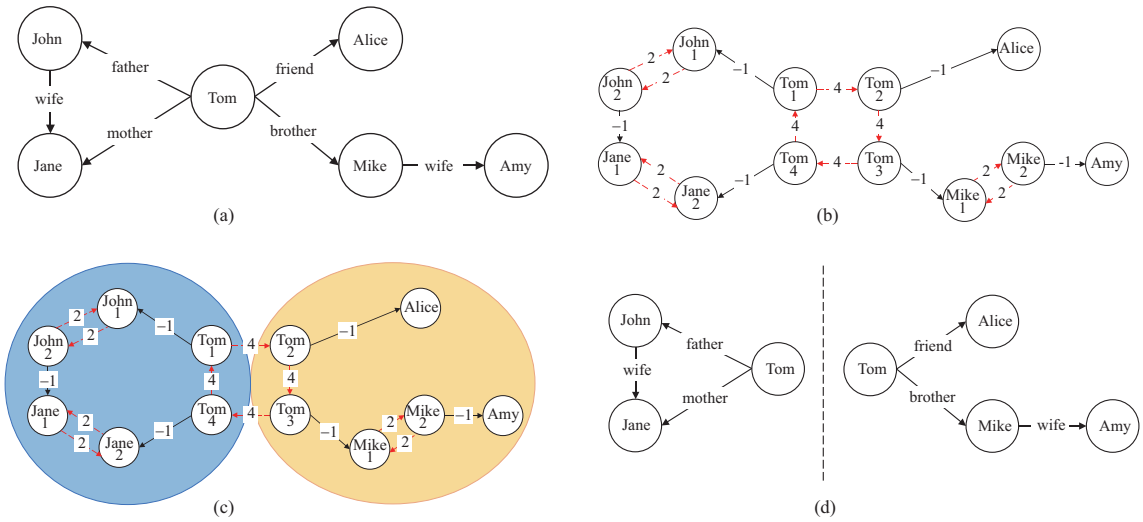


Figure 3 (Color online) Example of graph partition using G-SPAC. (a) Original graph; (b) transformed graph; (c) splitting; (d) result.

make it invisible to the entire graph. The subgraph division here extracts the corresponding results and converts them into the form of the original graph at the same time. This step will be described in detail in Subsection 6.3.

Step 3: regroup. Loop the above operations until the priority queue is empty. At this time, the remaining subgraphs that do not meet the limit are combined according to their size to form new subgraphs. So far, we have completed a general overview of the algorithm.

For example, there is a graph shown in Figure 3(a). We are going to split it into two parts with the G-SPAC algorithm. To begin with, the graph is transformed by copying the vertices for each edge. After copying the vertices, each of internal vertices from the same vertex is connected to its neighbors. Finally, every copy from the same vertex is joined together by sequence. Then, Figure 3(a) is successfully converted to Figure 3(b), on which we will conduct the graph split algorithm. The detail information will be described in Subsection 6.3. Figure 3(c) represents the splitting result, and it is restored to its original graph shown in Figure 3(d).

6 Detailed design

Our algorithm is designed to improve the SPAC algorithm with a greedy method according to the conclusion that cutting large-degree vertices is much more beneficial than cutting the small ones. In our paper,

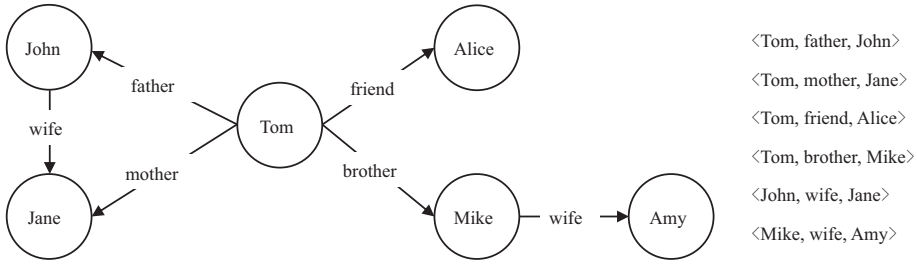


Figure 4 (Color online) Character relationship diagram of the example graph.

Algorithm 2 Greedy method.

Require: $\boldsymbol{E}'$;
Ensure: g: top vertex of the folded graph;
1: **while** $\boldsymbol{E}'$ **do**
2: // Select the edge with minimum weight;
3: minEdge $\leftarrow$ min($\boldsymbol{E}'$);
4: pop($\boldsymbol{E}'$, minEdge);
5: // Fold this edge and get its top vertex;
6: $g \leftarrow$ fold(minEdge);
7: **end while**
8: **return** g.

we gather the small-degree vertices to avoid letting them cut and thus reduce the number of vertices copies.

In the following, we will use the graph shown in Figure 4 as an example to explain the partition process. The graph shown in Figure 4 is a simple character relationship diagram showing Tom and his parents and friends. The right side of the graph is the edges that the graph contains, and they are exactly the input to the algorithm. After the algorithm obtains the graph, it first encodes the graph, encoding it sequentially into numbers. In order to show the division process intuitively, when using this example below, it will be explained in the form of characters. In this section, we will partition this graph into two parts.

6.1 Greedy method

The greedy strategy is an intuitive and simple but effective classical strategy. It does not pursue the global optimal solution but focuses on the current local optimal choice. It is hoped that a globally optimal solution or an approximate solution can be generated through step-by-step optimal selection. This paper embodies the greedy strategy in graph folding and shrinking. The edge with the smallest weight in the current edge set is selected for folding in each contraction. In this way, the edge of the vertex with the most minor degree is preferentially folded to gather the vertices with small degree into a group.

Algorithm 2 describes the above process. $\boldsymbol{E}'$ is the edge set to be folded, and the algorithm can return the top vertex of the folded group. The main part of the algorithm is a loop, which selects the minimum-weight edge minEdge of the current edge set each time and deletes it from the edge set. Then, fold the edge, get the top vertex of the folded group, and assign it to topVertex, which may be returned lately.

6.2 Transform

The transform stage mainly transforms the graph $\boldsymbol{G}$ to $\boldsymbol{G}'$ and adds it to the priority queue.

On the one hand, we are going to convert the graph $\boldsymbol{G}$, which is shown in Algorithm 3. To begin with, d_i copies of each vertex v_i are copied, where d_i is the degree of v_i. For edge $e = (v_i, v_j)$ in $\boldsymbol{G}$, there is edge $e' = (v_{im}, v_{jn})$ in $\boldsymbol{G}'$, where v_{im} represents the mth copy for vertex v_i, and v_{jn} represents the nth copy for vertex v_j. The edge e' is named the dominant edge, and its weight is -1. After connecting dominant edges, every copy from the same vertex v_i is joined together by sequence, which means that v_{ik} is connected to $v_{i(k+1)}$, and v_d is connected to v_0, where d is the degree of vertex v_i. We call the edge $e^i_{k,k+1}$ additional edge for v_i.

Taking Figure 4 as an example, according to the description, its converted image is shown in Figure 3(b). Its dominant edges are black, while additional edges are red for the convenience of distin-

Algorithm 3 TransformGraph.

Require: $\boldsymbol{G}$;
Ensure: $\boldsymbol{G'}$.
1: **for** $v_i \in \boldsymbol{V}$ **do**
2: // Copy d copies of v.
3: **for** $d--$ **do**
4: $v'_{ij} \leftarrow v_i$;
5: **end for**
6: // Join each vertices in $\boldsymbol{V'_i}$.
7: **for** $v'_{ij} \in \boldsymbol{V'_i}$ **do**
8: $e'_{i(j,(j+1)\%d)} \leftarrow (v'_{ij}, v'_{i((j+1)\%d)})$;
9: **end for**
10: // Join dominant edges.
11: **for** $e \in \boldsymbol{E}$ **do**
12: $e'_{im,jk} \leftarrow e_{i,j}$;
13: **end for**
14: **end for**
15: **return** $\boldsymbol{G'} \leftarrow (\boldsymbol{V'}, \boldsymbol{E'})$.

guishing. Compared to the graph shown in Figure 4, there are some subtle differences towards dominant edges. For instance, the origin edge $\langle \text{Tom}, \text{friend}, \text{Alice} \rangle$ is changed to $\langle \text{Tom2}, \text{friend}, \text{Alice} \rangle$ in the graph of Figure 3(b).

On the other hand, the degree-and-spatial-locality-prior method is performed with a priority queue. The priority queue is used to accomplish the design of greedy strategy, where the edges with small weights are before being processed. Besides, it allows more tightly folding since spatial locality is taken under consideration by the usage of the priority queue.

The priority queue is a kind of data structure that enables to sort data by priority. It is essentially a heap, whose physical implementation is an array while logical structure is a tree. The tree of a heap is a complete binary tree. Therefore, data could be stored in an array by level, and the position of data in the array could imply its logical position in the complete binary tree. In the tree, the parent node is always greater or smaller than its child nodes. That is, the greatest or the smallest node can be acquired at the root of a tree. As a result, we can get the data with the highest priority at the top of the priority queue. Meanwhile, the FIFO (first-in-first-out) feature of a queue is shown in the situation when manipulating data with the same priority. Specifically speaking, once there are two edges at the same rank, the first edge put into the queue will be the first one to dequeue. All in all, it could supply a stable method to sort data by priority.

In our paper, a priority queue is used to sort the additional edges. We select the essential messages of an edge to construct a light structure to make up the priority queue by the edges' weights. As we mentioned before, a priority queue is a heap in essence. Thus, it could provide an efficient sort for the additional edges. Besides, its FIFO feature contributes to sorting stability, and graphs are stored by region to a certain extent so that subgraphs will be folded more tightly. This is because the edge folding operation is always firstly performed on additional edges connecting to vertices copied from the same vertex in $\boldsymbol{G}$, and on its neighbor vertices in the second. That is to say, subgraphs tend to collapse locally and regionally.

Example 1. Take the graph shown in Figure 4 as an example, if we have a priority queue Q, it may contain edges flowing such order:
$\langle \text{Tom1}, -1, \text{John1} \rangle$, $\langle \text{Tom4}, -1, \text{Jane2} \rangle$,
$\langle \text{Tom2}, -1, \text{Alice} \rangle$, $\langle \text{Tom3}, -1, \text{Mike1} \rangle$,
$\langle \text{John2}, -1, \text{Jane1} \rangle$, $\langle \text{Mike2}, -1, \text{Amy} \rangle$,
$\langle \text{John1}, 2, \text{John2} \rangle$, $\langle \text{John2}, 2, \text{John1} \rangle$,
$\langle \text{Jane1}, 2, \text{Jane2} \rangle$, $\langle \text{Jane2}, 2, \text{Jane1} \rangle$,
$\langle \text{Mike1}, 2, \text{Mike2} \rangle$, $\langle \text{Mike2}, 2, \text{Mike1} \rangle$,
$\langle \text{Tom1}, 4, \text{Tom2} \rangle$, $\langle \text{Tom2}, 4, \text{Tom3} \rangle$,
$\langle \text{Tom3}, 4, \text{Tom4} \rangle$, $\langle \text{Tom4}, 4, \text{Tom1} \rangle$.

6.3 Judgment-aware folding

In this subsection, judgment-aware folding is proposed to alleviate the division imbalance, allowing partitioning graphs in a more granular perspective.

Algorithm 4 Folding.

Require: e, arr;
Ensure: u: the top vertex of the current subgraph the edge e belongs to.
1: $u \leftarrow$ GetTopVertex($e.u$);
2: $v \leftarrow$ GetTopVertex($e.v$);
3: arr[v][0] $\leftarrow u$;
4: arr[v].push(u);
5: Update relevant parameters;
6: **return** u.

Algorithm 5 GetTopVertex.

Require: v, arr;
Ensure: The top vertex that contains v.
1: **while** arr[v][0] $\neq v$ **do**
2: $v \leftarrow$ arr[v];
3: **end while**
4: **return** v.

6.3.1 *Edge folding*

When it comes to the concept of folding edges, METIS [41] must be mentioned, which was used as the vertex partition method on the converted graph in [39]. Indeed, METIS [41] is a classical algorithm with excellent performance. In our paper, we provide another usage of edge folding with a finer perspective, which is easier to implement yet satisfactory.

Conceptually, folding edges means one vertex overwrites the other vertex connected to it. Therefore, the two vertices are combined together, and the edge they connect becomes the inside edge. To provide a practical method, the folding edge has three meanings in our paper. Let $e = (u, v)$ represent the edge to be folded. Firstly, one vertex overwrites the other vertex, which means they share one vertex together. For simplicity, we do not generate a new vertex to represent the combination of the vertices but hide one vertex under the other. For example, vertex v is covered under vertex u, so that v is invisible to the rest of the graph. Furthermore, this implies that the top vertex actually contains a subgraph. Nevertheless, vertex v could contain a subgraph, and the covering operation joins the subgraph to the one u contains. Secondly, the folded edge becomes the inside edge, which means it is invisible to the rest of the graph. Thirdly, the rest of edges connected to v will be virtually connected to u, since v is invisible and u is the representation of v.

Just to be clear, not every edge will get folded, such as an edge connected to a subgraph that has already been split away or the edge between two subgraphs which are contained in the same vertex. In a word, only the edge that joins two disconnected existing subgraphs will be folded. If a subgraph is split away, it is regarded as nonexistence.

The process of folding an edge is given by the pseudocode shown in Algorithm 4. The algorithm requires the edge $e = (u, v)$ which is to be folded and an array arr which records the messages needed for folding. The algorithm returns the top vertex representing the collapsed subgraph. First, GetTopVertex function is called to get the top vertices of the two endpoints of this edge, as shown in Algorithm 5. Second, update the values of the two top vertices in arr and other relevant information. Finally, the top vertex representing the collapsed subgraph is returned.

We are going to fold the graph of Figure 3(b) as an example. The graph shown in Figure 5 is what it looks like after all the dominant edges are folded. The number in parentheses in the figure is the number of dominant edges actually contained in the subgraph represented by the vertex. According to the sequence in the priority queue Q, the next edge to be folded is $\langle \text{John1}, 2, \text{John2} \rangle$, which could be virtually presented as $\langle \text{Tom1}, 2, \text{John2} \rangle$ in Figure 6. As for the next edge $\langle \text{John2}, 2, \text{John1} \rangle$, its two ends are contained within subgraph Tom1, so there is no need to fold it. Then come to the edge $\langle \text{Jane1}, 2, \text{Jane2} \rangle$, which connects vertices John2 and Tom4. After combining these two subgraphs, shown in Figure 7, we have the subgraph that meets the limit L and can split it away. At last, we have the subgraphs needed shown in Figure 3(d).

6.3.2 *Limit judgment*

After folding an edge, the new subgraph may be split away. The process of division is shown in the pseudocode given by Algorithm 6. The function needs the top vertex v representing the subgraph to be divided and an array arr recording the folding information. It mainly loops through the following steps. Step 1, the edge at the row of v in arr is obtained. Step 2, determine whether the edge is dominant, and

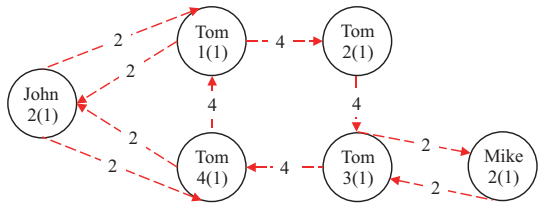


Figure 5 (Color online) Folding dominant edges of example.

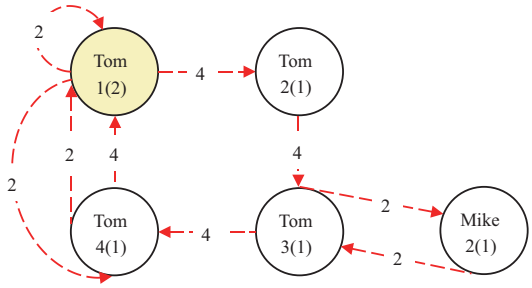


Figure 6 (Color online) Folding edge ⟨John1, 2, John2⟩.

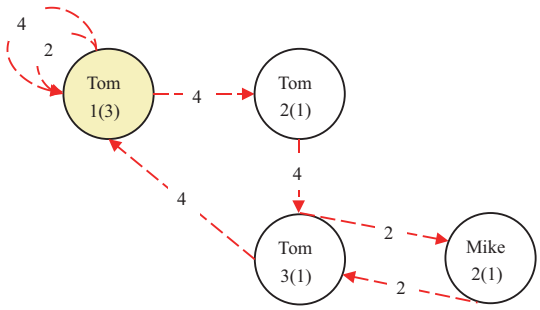


Figure 7 (Color online) Folding edge ⟨Jane1, 2, Jane2⟩.

Algorithm 6 Split.

Require: v, arr;
Ensure: file of subgraph.
1: **for** $i \leftarrow 1$ to arr[v].size **do**
2: $e \leftarrow (v, \text{arr}[v][i])$;
3: **if** e.weight $== -1$ **then**
4: res.push(e);
5: **end if**
6: **end for**
7: **return** file $\leftarrow$ res.

if so, add it to the result set. Having handled all the edges related to v, the result set is stored in a file and returned.

Limit is set to judge if the subgraph should be split away once the edge is folded. To achieve a balanced partition, we hope that a total number of edges can be partitioned evenly. For example, there are k machines storing the graph $\boldsymbol{G}$ together. Then, each machine is expected to contain exactly m/k edges, where m is the number of edges in $\boldsymbol{G}$. That is, the limit can be represented as m/k.

Practically, since we perform vertex partition on the converted graph $\boldsymbol{G}'$, the sum of vertices collected is intuitive to achieve. According to the algorithm, after folding domain edges, every domain edge equals two vertices. Thus, during the following steps, the sum of folded domain edges is half of the one of vertices. Therefore, the number of vertices could be used to match the limit. Let g'_i be the ith subgraph of g', and it contains n'_i vertices in total.

As mentioned in Subsection 6.3.1, we split the subgraph while folding edges. It inspires us to reset the limit for the rest of the subgraphs. In other words, the limitation value will be recalculated according to the remaining edges and machines. This helps to keep balance when the previous split subgraph is excessive. In this case, once the limit is not adjusted, it would further lead to unbalanced division since there are not enough edges for the rest to divide according to the old limit. On the contrary, resetting the limit while splitting provides a finer control of subgraph size.

6.4 Time complexity

Assume that a graph is defined as $\boldsymbol{G} = (\boldsymbol{V}, \boldsymbol{E})$, in which $\boldsymbol{V}$ is the vertex set, and $\boldsymbol{E}$ is the edge set. Let m represent the number of edges in $\boldsymbol{E}$. Then, $\boldsymbol{G}' = (\boldsymbol{V}', \boldsymbol{E}')$ refers to the graph transformed from $\boldsymbol{G}$. In the new graph $\boldsymbol{G}'$, let m' represent the size of the edge set $\boldsymbol{E}'$.

Table 1 Summary of datasets, where most of the information is quoted from [43].

Data set	Edges	Vertices	Density	Maximum degree	Minimum degree	Average degree
in-2004	16.9M	1.4M	1.73×10^{-5}	2.19×10^{4}	0	23
soc-FourSquare	3.2M	639.0K	1.57×10^{-5}	1.06×10^{5}	1	10
scircuit	958.9K	171.0K	5.39×10^{-5}	704	0	9
cant	2.0M	62.5K	1.02×10^{-3}	77	18	63
mc2depi	2.1M	525.8K	1.14×10^{-5}	6	2	5
C2000-5	999.8K	2K	5.00×10^{-1}	1.1×10^{3}	919	999
C1000-9	450.1K	1K	9.01×10^{-1}	925	868	900
frb45-21-4	387.5K	945	8.69×10^{-1}	875	732	820
brock200-1	14.8K	200	7.45×10^{-1}	165	130	148
c-fat200-2	3.2K	200	1.62×10^{-1}	34	32	32
DSJC500-5	62.6K	500	5.02×10^{-1}	286	220	250
hamming6-4	704	64	3.49×10^{-1}	22	22	22
johnson8-4-4	1.9K	70	7.68×10^{-1}	53	53	53
keller4	9.4K	171	6.49×10^{-1}	124	102	110
MANN-a27	70.6K	378	9.90×10^{-1}	374	364	373
p-hat700-1	61K	700	2.49×10^{-1}	286	75	174
san200-0-7-2	13.9K	200	7.00×10^{-1}	164	103	139
gen400-p0-9-65	71.8K	400	9.00×10^{-1}	378	333	359

In this subsection, the time complexity of the G-SPAC algorithm will be analyzed. For briefness, it is assumed that when graph $\boldsymbol{G}$ is partitioned in balance, graph $\boldsymbol{G}'$ is also partitioned in balance. During the analysis process, the folding operation is selected as a basic operation. Each folding operation needs to find the outermost vertex of the current vertex. In the worst case, for each subgraph, the number of operations required for this process increases from 1 to $m'/k-1$, that is, the total number of operations is to be $m'(m'/k-1)/(2k)$. Then, for k subgraphs, it needs to perform the folding operation in $m'^2/(2k)-m'/2$ times. Therefore, for a fixed k, the time complexity of the algorithm is m'^2.

7 Experiment evaluation

In this section, experiments will be conducted to evaluate the performance of the G-SPAC algorithm from different perspectives. In these experiments, the number of partitions to be divided is set to 3, which is the least amount of computing nodes in a distributed system. Performance with different partition numbers is also examined.

7.1 Data sets

As shown in Table 1 [43], our experiments are carried out on 18 data sets, where in-2004, scircuit, cant, and mc2depi are in the category of real-world sparse graphs [44] and soc-FourSquare social networks [43], frb45-21-4 is in the category of BHOSLIB, and the remaining data sets are from DIMACS. The sizes of the tested datasets vary from hundreds of thousands to tens of millions, and there are four kinds of graphs among them, including power-law graph, inverse power-law graph, mesh-like graph, and normal distribution graph.

As mentioned earlier, the power-law graph is a kind of graph in which most vertices have a few neighbor vertices while very few vertices have a great degree. This paper presents power-law graphs, such as data sets in-2004, soc-FourSquare, and scircuit, whose probability cumulative curves have been given in Figure 8(a). We can see that the curve of in-2004 is the flattest one in relative terms. And its average degree is about twice that of the other two graphs. The curve next to it belongs to the scircuit, whose difference between the minimum degree and maximum degree is the least one. But the density of the scircuit is about twice that of the others. As for soc-FourSquare, according to Table 1, it has the largest gap of vertex degrees among the data sets varying from 1 to more than 100 thousand. However, its average degree is only 10, which confirms the power law feature.

Compared with the power-law graph, the other graphs seem much more even. For example, unlike the graphs above, cant, represented in Figure 8(b), shows exact opposite characteristics — most vertices have large degrees while a few have small degrees. In addition, looking over Figure 8(c), we can find

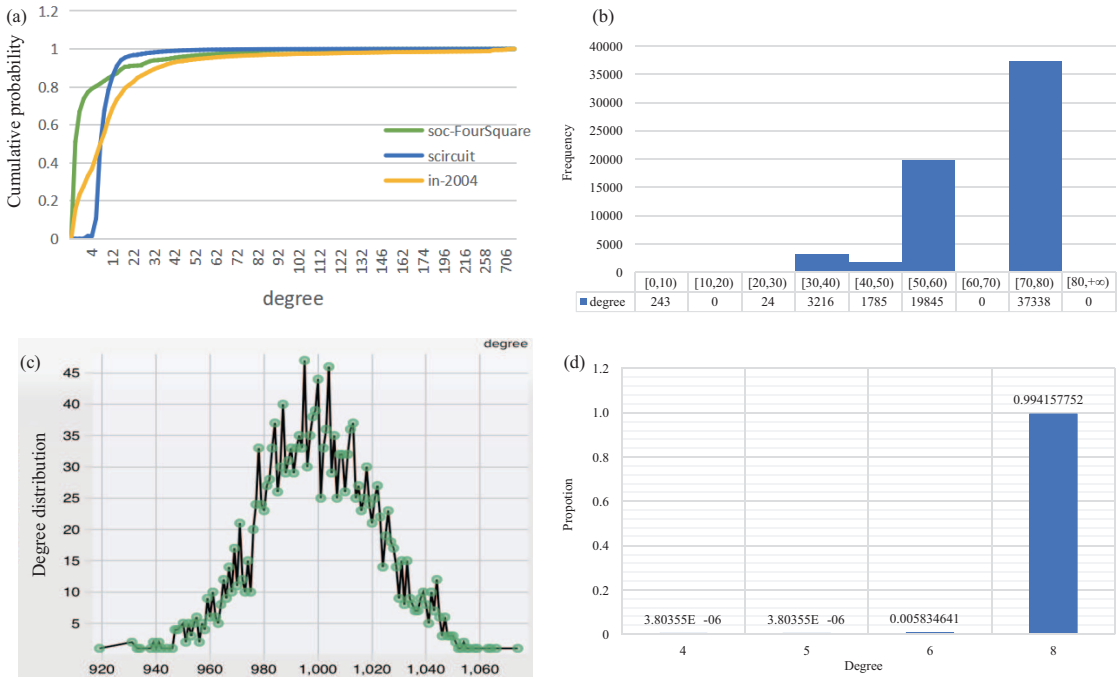


Figure 8 (Color online) Degree distributions of adopted graphs. (a) Scircuit and soc-FourSquare; (b) cant; (c) C2000-5; (d) mc2depi.

Table 2 Comparison of the vertex copies of G-SPAC and SPAC.

Graph	SPAC	G-SPAC	Graph	SPAC	G-SPAC
mc2depi	36124	4569	c-fat200-2	–	238
cant	49708	57034	DSJC500-5	–	813
scircuit	5607	73202	hamming6-4	–	94
in-2004	199881	14806	johnson8-4-4	–	107
soc-FourSquare	–	105332	keller4	–	281
C2000-5	–	3296	MANN-a27	–	652
C1000-9	–	1633	p-hat700-1	–	1009
frb45-21-4	–	1539	san200-0-7-2	–	303
brock200-1	–	329	gen400-p0-9-65	–	406

that C2000-5 takes on another kind of degree distribution — the degree of medium size accounts for the majority. Meanwhile, its density is relatively high. Referring to the mesh-like graph — mc2depi, represented in Figure 8(d), contributes to the least gap of degree and the highest balance.

The following experiments are performed on the g++ 7.5.0 with Ubuntu 18.04.4 operating system. The efficiency and balance of division are respectively measured by using the number of copied vertices and relative standard deviations.

7.2 Vertex copies

In this subsection, we compare the cost of vertex copies defined in (1) to SPAC algorithm [39], listing data at (Table 2). Four original data sets were tested to contrast with data from [39], and other data sets were added to validate G-SPAC's performance further. Figure 9(a) is the percentage accumulative histogram, showing the difference in experiment results between two algorithms on the original data sets. Broadly speaking, our G-SPAC algorithm provides an alternative to the SPAC algorithm on some graphs and performs much better on other graphs.

Figure 9(a) describes the difference of vertex copies when compared with the SPAC algorithm. It is clear that on cant datasets, the boundary of two algorithms is around 50%, meaning that they copy about the same amount of vertex when partitioning this graph. Moreover, the comparison bars of data sets mc2depi and in-2004 fall into the distinct shape, where our G-SPAC algorithm only takes one-tenth of

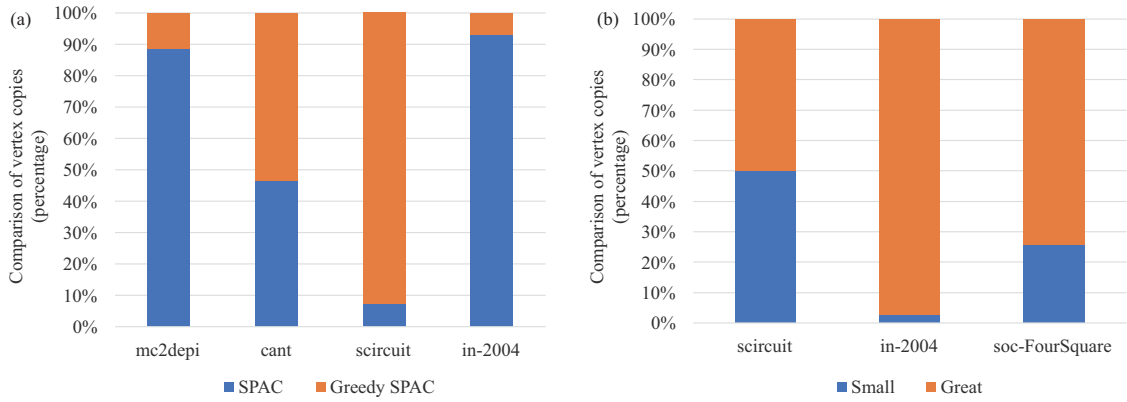


Figure 9 (Color online) (a) Cost of vertex copies; (b) small degree prior and great degree prior.

vertex copies needed for the SPAC algorithm to partition the graphs, which is totally reduced by order of magnitude. However, G-SPAC does not perform well in the data set scircuit when compared with SPAC.

scircuit and in-2004 are both power-law graphs, but their results vary considerably. We assume that this is because of the proportion of small-degree vertices. From Figure 8(a) we can know that the probability cumulative curve of scircuit is abrupter than that of in-2004, which means the proportion of small-degree vertices in scircuit is greater. It leads to a lower probability of splitting the large-degree vertices, which is against the design intention of our algorithm – splitting the large-degree vertices is more efficient. So compared with scircuit, in-2004 creates better quality.

We also noticed that, for data set mc2depi, the cost of copying vertex of the G-SPAC algorithm is much less than that of SPAC, accounting for only one-tenth. Its degree proportion distribution, given in Figure 8(d), demonstrates mc2depi is a simple and balanced mesh-like graph. The key point to partitioning such graphs is accumulating subgraphs locally. Generally, graphs are stored by spatial sequence to some extent, whereas neighbor vertices are usually coded closely. The priority queue could employ this feature to ensure the vertex sequence of subgraphs.

We noticed the high density of C2000-5 and subsequent data sets in accordance with Table 1. And the high-density means vertices are very closely connected. In the algorithm, the edges connected by the smaller vertices are folded under the remaining major vertices before cutting the larger vertices. For a graph with distinct sizes and vertices, such as a power-law graph, cutting a large vertex graph can effectively separate the connected small vertex sets. However, cutting any large vertex is ineffective for a dense graph. So, the cost of copying the number of vertices is also large.

The cant graph has a similar problem. It is gathered in small units, and each small group has a regular degree distribution, similar to 20, 20, 50, and 50. For such graphs, the best partitioning would be in small groups, but our algorithm does not handle it that way. The additional edges of vertices with smaller degrees are preferentially folded, resulting in different degrees of fusion between different small groups, blurring their regularity. Therefore, the number of replicated vertices consumed by partitioning the cant graph is not significantly reduced by using the greedy strategy.

Furthermore, we compare the difference between the G-SPAC algorithms with priority to fold small weighted edges and that with priority to fold great weighted edges, showing results in Figure 9(b). Especially, results from power-law graphs are particularly picked out to have a comparative analysis with percentage accumulative histogram at Figure 9(b). We can see that sorting edges with small-weighted priority costs less than that with great-weighted priority. The latter method tends to gather vertices with great degrees and split vertices with small degrees, verifying the efficiency improvement by splitting large-degree vertices.

7.3 Partition balance

In this subsection, partition balance is examined by looking over the relative standard deviation (RSD) of partitions. Results for each dataset are given in Figure 10. For comparison, the partition unbalance is less than 1% and 3% using different strategies in METIS [41], which is used as a case in SPAC. It is necessary to point out that there is no evidence to indicate that the definition of partition balance in METIS also applies to SPAC.

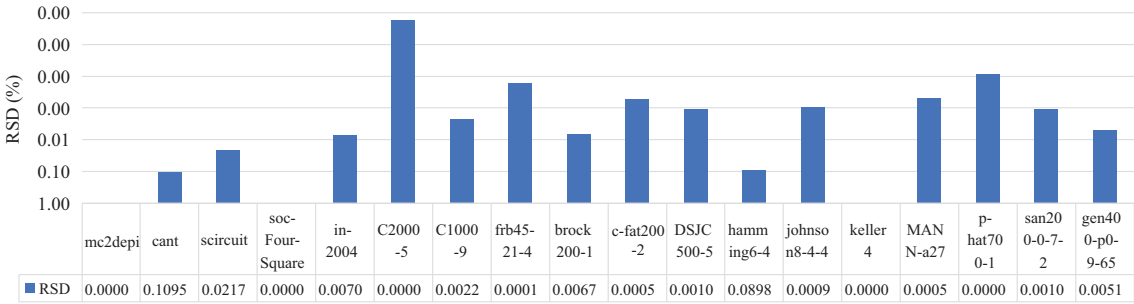


Figure 10 (Color online) Partition balance (RSD) of G-SPAC.

In accordance with Figure 10, it is apparent that there are two entirely different outcomes. Firstly, to most of the data sets, the RSD is less than 1%. Particularly, some of the graphs could be exactly divided into k partitions equally. Due to the previous results, our algorithm has excellent performance dealing with evenly distributed graphs, such as mc2depi, both in terms of algorithm cost and partition balance. This is because we use the priority queue data structure, combined with the characteristics of local aggregation on the data set, so as to achieve continuous segmentation of subgraphs.

Another kind of graphs whose degree distribution tends to be balanced, such as C2000-5, C1000-9, and frb45-21-4. It can be seen from Figure 8(c) that, unlike the grid-like balance graph shown by mc2depi, the difference between the maximum vertex degree and the minimum vertex degree of each of the three graphs is about 100, which is relatively small compared to the overall vertex degrees, so it can be considered that this data set is relatively balanced. The partition balance of smoothly distributed graphs, as stated above, can be supported in our algorithm.

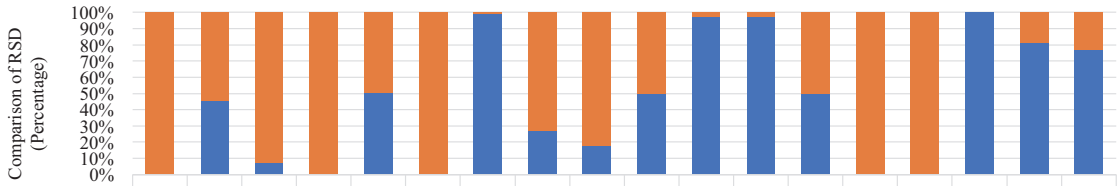
The power-law graphs used in the experiment can be divided into two types. One has an extremely unbalanced distribution, which is represented by in-2004 and soc-FourSquare. They are characterized by distinct clusters, outside of which the vertices are scattered. This kind of power-law graph has excellent partition balance. The other is distributed in small units, as shown by scircuit. When this kind of graph is divided, it is easy to merge adjacent small unit groups, which leads to the rapid growth of the size of the subgraph, and thus the division is relatively unbalanced. But still, its RSD is under 3%.

Similar problems are also reflected in the division of cant and hamming6-4. They are clustered in small units, and each small group has a regular degree distribution. The result of this small collective distribution is that they are more inclined to merge into collective units, which is not conducive to achieving the purpose of a balanced graph division.

7.4 Spatial locality

Next, the experiment is conducted on whether or not the priority queue is used to test the performance of spatial locality. In other words, it tests the performance if the additional edges from the same vertex could be partitioned together as far as possible. In the experiment, the control group uses the vector data structure and uses the sort function to sort them from small to large. Since the sort function is unstable, it would disorder the edges with the same rank regardless of whether they are from the same vertex. In this way, the folding phase may be much more disorganized. We compare the data set partition balance, RSD, of the two methods. The comparison results are shown in Figure 11, which is a percentage-stacked column chart that visually shows the degree of difference.

From Figure 11, we can see that using a priority queue is not inferior or even superior to using the sort function on the vector data structure. Except for the data sets mc2depi, scircuit, and hamming6-4, the priority queue has an overwhelming advantage. Although, from Figure 11, it appears that there are huge differences in the results of dividing data sets soc-FourSquare, C2000-5, C1000-9, frb45-21-4, DSJC500-5, hamming6-4, p-hat700-1, san200-0-7-2, and gen400-p0-9-65 between the two methods, actually the values of RSD are far below 1% (expect for hamming6-4, which performs the RSD of around 9%, which is also small). In particular, for the data sets mc2depi, keller4, and MANN-a27, the RSD of the division by the priority queue method is 0, while another method performs relatively large values of RSD (around 50% for mc2depi, 4% for keller, and 17% for MANN-a27). These data sets are similar to the mesh-like map in scientific research, where the distribution is simple and relatively uniform, and it only needs to be divided by regions to obtain good division results. One of the characteristics of priority queues is that



	mc2d epi	cant	scircuit	soc-Four Square	in-2004	C2000-5	C1000-9	frb45-21-4	brock200-1	c-fat200-2	DSJC500-5	hamming6-4	johnson8-4-4	keller4	MANN-a27	p-hat700-1	san2000-0-7-2	gen400-p0-9-65
Vector	0.4861	0.1286	0.2821	0.0000	0.0070	0.0010	0.0000	0.0004	0.0306	0.0005	0.0000	0.0024	0.0009	0.0410	0.1700	0.0000	0.0002	0.0015
Priority Queue	0.0000	0.1095	0.0217	0.0000	0.0070	0.0000	0.0022	0.0001	0.0067	0.0005	0.0010	0.0898	0.0009	0.0000	0.0005	0.0000	0.0010	0.0051

Figure 11 (Color online) Comparison of partition balance (RSD) between G-SPAC without priority queue and G-SPAC.

data is stored in spatial order, so the priority queue can be very suitable for the grid division of scientific research.

Furthermore, we found that both methods have poor partition balance when splitting graph scircuit. As has been analyzed before, the data set scircuit is cluster-based, and the scales between different clusters are similar. For such distributed graphs, greedy strategies tend to break their regularity, resulting in the mixing of clusters. Combining connected subgraphs makes it easy to perform subgraph segmentation operations without fully including the cluster, that is, splitting small groups, which is not conducive to balanced division.

7.5 Limit judgment

The difference between using dynamic/static limits is compared in this subsection. As explained in the previous Subsubsection 6.3.2, the limit can be obtained by dividing the number of edges in the original graph by the number of partitions. If this limit remains constant during the partitioning process, it is called a static decision. Conversely, if the limit is adjusted after each division, it is called a dynamic decision. These two judgment methods are compared in the above 18 data sets. The results shown in Figures 12(a) and (b) reflect that in most cases, the dynamic approach is superior to the static one on both costs of vertex copies and partition balance, except for the case of dealing with data set in-2004.

It is distinct that these two judgment methods have totally different performance for partitioning balance and vertex copies. Especially for in-2004, the static judgment method contributes to a perfect balance of partitioning while a large number of vertex copies are required. In addition, as shown in Figure 12(b), though it seems the inferior balance of the dynamic method for in-2004, brock200-1, and hamming6-4, actually the RSD values of the dynamic method for in-2004 and brock200-1 are far below 1%, and the RSD value of the dynamic method is only slightly higher than that of the static method for hamming6-4. All in all, it can be seen from the results that the dynamic method has a better division balance than the static method, and the number of vertices copied by the dynamic method is less than that of the static method.

The advantage of the dynamic approach is that the division limits can be readjusted to make the remainder more balanced under most conditions. This determination method is especially advantageous for situations where there is a division imbalance.

7.6 Partition numbers

This subsection will discuss the impact of different numbers of partition nodes on partition performance. The experiment will make a comprehensive comparison from the number of replicated vertices and partition balance. Experiments show that for most datasets, the number of vertices required to replicate increases with the number of partition nodes. And the partition balance is almost the same for most of them.

As shown in Figures 13(a) and (b), overall, the number of replicated vertices increases with the number of partition nodes. This is because more partition nodes usually mean more connection points are needed to connect the individual subgraphs into a complete graph. That is, more vertices are copied. On the contrary, for the graph scircuit, when the number of partition nodes increases, the number of vertices required to be copied is greatly reduced. However, it comes at the expense of a decrease in partition

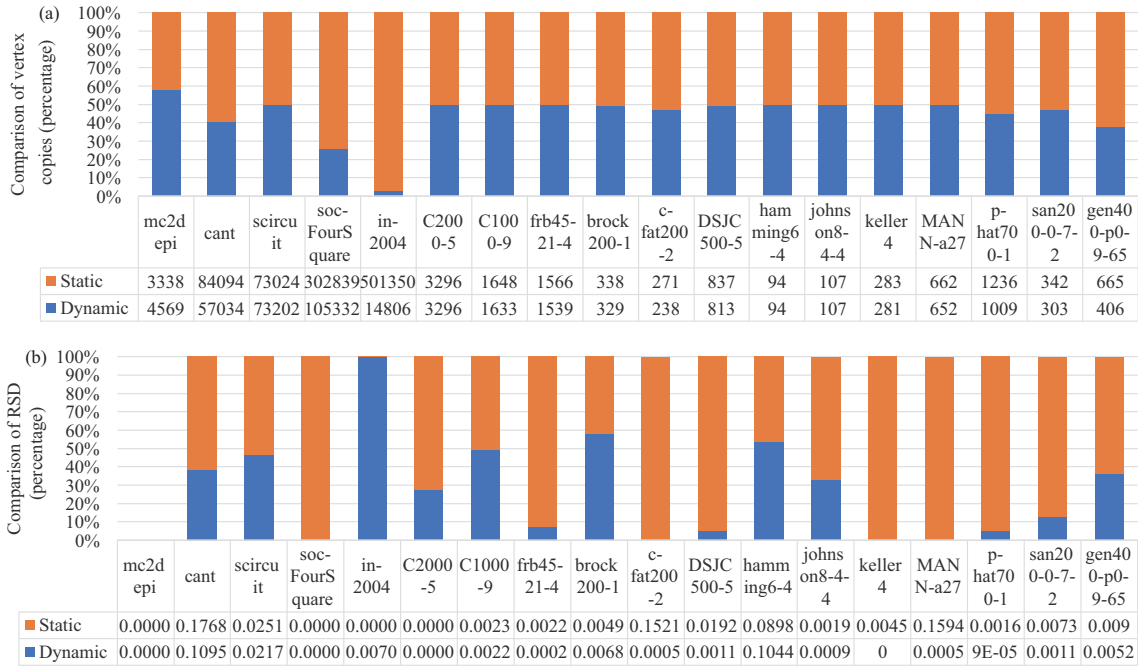


Figure 12 (Color online) (a) Cost of vertex copies by two decisions; (b) partition balance by two decisions.

Figure 13 (Color online) (a) Cost of vertex copies with different numbers of partitions; (b) RSD with different numbers of partitions.

balance. In comparison, datasets such as mc2depi, soc-FourSqure, and in-2004 have better stability, performing well under different numbers of partition nodes.

8 Conclusion

In this paper, we use the greedy strategy to improve the SPAC algorithm and compare the partition balance and partition efficiency on six data sets. The results of experiments show that, in general, the G-SPAC algorithm is not inferior to the SPAC algorithm in terms of the number of replicated vertices. In addition, we also tested the partition balance of the algorithm. For the vast majority of data sets, the division balance is less than 3%, while the cant data set has poor division balance, which is related to its

distribution. In particular, for mesh-like graphs such as mc2depi, the G-SPAC algorithm has excellent performance. It not only has a high division balance but also has far fewer replicated vertices than the SPAC algorithm.

At present, the processing of graphs tends to be parallelized, which can greatly reduce the division time. Therefore, the next step in improving our algorithm is to parallelize it. And the difficulty lies in realizing a greedy approach in parallelization with a priority queue.

Acknowledgements This work was partially supported by National Natural Science Foundation of China (Grant Nos. 62172157, 62202149), Key R&D Program of Hunan Province (Grant No. 2023GK2002), and Programs of Shenzhen and Guangdong Province (Grant Nos. 2023A1515012915, JCYJ20210324135409026). The authors would like to thank the reviewers for their invaluable and helpful comments and revision suggestions on improving the manuscript.

References

1 Myers S A, Sharma A, Gupta P, et al. Information network or social network?: the structure of the Twitter follow graph. In: Proceedings of the 23rd International World Wide Web Conference, Seoul, 2014. 493–498
2 Wang T Y, Chen Y, Zhang Z B, et al. Understanding graph sampling algorithms for social network analysis. In: Proceedings of the 31st IEEE International Conference on Distributed Computing Systems Workshops (ICDCS 2011 Workshops), Minneapolis, 2011. 123–128
3 Shuang K, Su S. PAIDD: a hybrid P2P-based architecture for improving data distribution in social networks. Sci China Inf Sci, 2014, 57: 042309
4 Shin Y Y, Yoon Y. Incorporating dynamicity of transportation network with multi-weight traffic graph convolutional network for traffic forecasting. IEEE Trans Intell Transp Syst, 2022, 23: 2082–2092
5 Wehmuth K, Costa B, Bechara J V, et al. A multilayer and time-varying structural analysis of the Brazilian air transportation network. In: Proceedings of the Latin America Data Science Workshop co-located with 44th International Conference on Very Large Data Bases (VLDB 2018), Rio de Janeiro, 2018. 57–64
6 Yu Y F, Xu G X, Jiang M, et al. Joint transformation learning via the $L_{2,1}$-norm metric for robust graph matching. IEEE Trans Cybern, 2021, 51: 521–533
7 Peng X, Yu Z D, Yi Z, et al. Constructing the L_2-graph for robust subspace learning and subspace clustering. IEEE Trans Cybern, 2017, 47: 1053–1066
8 Wu X M, Du M N, Chen W H, et al. Salient object detection via region contrast and graph regularization. Sci China Inf Sci, 2016, 59: 032104
9 Xiao H, Chen Y D, Shi X D. Knowledge graph embedding based on multi-view clustering framework. IEEE Trans Knowl Data Eng, 2021, 33: 585–596
10 Shen Y, Ding N, Zheng H T, et al. Modeling relation paths for knowledge graph completion. IEEE Trans Knowl Data Eng, 2021, 33: 3607–3617
11 Zhang L L, Li D W, Xi Y G, et al. Reinforcement learning with actor-critic for knowledge graph reasoning. Sci China Inf Sci, 2020, 63: 169101
12 Mayer C, Tariq M A, Mayer R, et al. GrapH: traffic-aware graph processing. IEEE Trans Parallel Distrib Syst, 2018, 29: 1289–1302
13 Meyerhenke H, Sanders P, Schulz C. Parallel graph partitioning for complex networks. IEEE Trans Parallel Distrib Syst, 2017, 28: 2625–2638
14 Zhang Y, Liao X F, Jin H, et al. HotGraph: efficient asynchronous processing for real-world graphs. IEEE Trans Comput, 2017, 66: 799–809
15 Zhang W D, Zhang M Y. Graph partitioning algorithm with LSH: poster extended abstract. In: Proceedings of the IEEE International Conference on Cluster Computing, Belfast, 2018. 166–167
16 Delling D, Goldberg A V, Razenshteyn I P, et al. Graph partitioning with natural cuts. In: Proceedings of the 25th IEEE International Symposium on Parallel and Distributed Processing, Anchorage, 2011. 1135–1146
17 Vikas N. Computational complexity of graph partition under vertex-compaction to an irreflexive hexagon. In: Proceedings of the 42nd International Symposium on Mathematical Foundations of Computer Science, Aalborg, 2017
18 McCrabb A, Bertacco V. Optimizing vertex pressure dynamic graph partitioning in many-core systems. IEEE Trans Comput, 2021, 70: 936–949
19 Soudani N M, Fatemi A, Nematbakhsh M. PPR-partitioning: a distributed graph partitioning algorithm based on the personalized PageRank vectors in vertex-centric systems. Knowl Inf Syst, 2019, 61: 847–871
20 Jiang H, Zhu D M, Xie Z C, et al. A new upper bound based on vertex partitioning for the maximum k-plex problem. In: Proceedings of the 30th International Joint Conference on Artificial Intelligence, 2021. 1689–1696
21 Kwok T C, Lau L C, Lee Y T. Improved Cheeger's inequality and analysis of local graph partitioning using vertex expansion and expansion profile. SIAM J Comput, 2017, 46: 890–910
22 Zhang C Z, Wei F, Liu Q, et al. Graph edge partitioning via neighborhood heuristic. In: Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Halifax, 2017. 605–614
23 Ayall T, Duan H C, Liu C H, et al. Taking heuristic based graph edge partitioning one step ahead via offstream partitioning approach. In: Proceedings of the 37th IEEE International Conference on Data Engineering, Chania, 2021. 2081–2086
24 Ji S W, Bu C Y, Li L, et al. Local graph edge partitioning with a two-stage heuristic method. In: Proceedings of the 39th IEEE International Conference on Distributed Computing Systems, Dallas, 2019. 228–237
25 Li H, Yuan H, Huang J. Real-time edge repartitioning for dynamic graph. In: Proceedings of the 28th ACM International Conference on Information and Knowledge Management, Beijing, 2019. 2125–2128
26 Mayer R, Jacobsen H. Hybrid edge partitioner: partitioning large power-law graphs under memory constraints. In: Proceedings of the International Conference on Management of Data, 2021. 1289–1302
27 Zhang S, Jiang Z T, Hou X Z, et al. An efficient and balanced graph partition algorithm for the subgraph-centric programming model on large-scale power-law graphs. In: Proceedings of the 41st IEEE International Conference on Distributed Computing Systems, Washington DC, 2021. 68–78
28 Wang J Y, Zhang C F. Analysis and evaluation of the GAS model for distributed graph computation. In: Proceedings of the 37th IEEE International Conference on Distributed Computing Systems Workshops, Atlanta, 2017. 283–285
29 Zhou A C, Ibrahim S, He B S. On achieving efficient data transfer for graph processing in geo-distributed datacenters. In: Proceedings of the 37th IEEE International Conference on Distributed Computing Systems, Atlanta, 2017. 1397–1407
30 Sheng F, Cao Q, Jiang H, et al. GraBi: communication-efficient and workload-balanced partitioning for bipartite graphs. In: Proceedings of the 49th International Conference on Parallel Processing, Edmonton, 2020
31 Miao H, Liu X Y, Huang B, et al. A hypergraph-partitioned vertex programming approach for large-scale consensus optimization. In: Proceedings of the IEEE International Conference on Big Data (IEEE BigData 2013), Santa Clara, 2013. 563–568

32 Bourse F, Lelarge M, Vojnovic M. Balanced graph edge partition. In: Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, New York, 2014. 1456–1465
33 Mayer C, Mayer R, Tariq M A, et al. ADWISE: adaptive window-based streaming edge partitioning for high-speed graph processing. In: Proceedings of the 38th IEEE International Conference on Distributed Computing Systems, Vienna, 2018. 685–695
34 Meyerhenke H, Sanders P, Schulz C. Parallel graph partitioning for complex networks. In: Proceedings of the IEEE International Parallel and Distributed Processing Symposium, Hyderabad, 2015. 1055–1064
35 Xie C, Yan L, Li W J, et al. Distributed power-law graph computing: Theoretical and empirical analysis. In: Proceedings of the Advances in Neural Information Processing Systems 27, Montreal, 2014. 1673–1681
36 Suri S, Vassilvitskii S. Counting triangles and the curse of the last reducer. In: Proceedings of the 20th International Conference on World Wide Web, Hyderabad, 2011. 607–614
37 Gonzalez J E, Low Y, Gu H J, et al. PowerGraph: distributed graph-parallel computation on natural graphs. In: Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation, Hollywood, 2012. 17–30
38 Abou-Rjeili A, Karypis G. Multilevel algorithms for partitioning power-law graphs. In: Proceedings of the 20th International Parallel and Distributed Processing Symposium (IPDPS 2006), Rhodes Island, 2006
39 Li L D, Geda R, Hayes A B, et al. A simple yet effective balanced edge partition model for parallel computing. Proc ACM Meas Anal Comput Syst, 2017, 1: 1–21
40 Hendrickson B, Kolda T G. Graph partitioning models for parallel computing. Parallel Computing, 2000, 26: 1519–1534
41 Karypis G, Kumar V. Metis-unstructured graph partitioning and sparse matrix ordering system, version 2.0. Appl Phys Lett, 1995, 97: 124101
42 Fan W F, Xu J B, Wu Y H, et al. Parallelizing sequential graph computations. In: Proceedings of the ACM International Conference on Management of Data, Chicago, 2017. 495–510
43 Rossi R A, Ahmed N K. The network data repository with interactive graph analytics and visualization. In: Proceedings of the 29th AAAI Conference on Artificial Intelligence, Austin, 2015. 4292–4293
44 Davis T A, Hu Y F. The University of Florida sparse matrix collection. ACM Trans Math Softw, 2011, 38: 1–25